**Artificial Intelligence for Developing Generative AI Applications**

**CSA – 1751**

**Shaik Nayaz Irfan Ali – 192211326**

**1.Write a Python program to solve 8 queens’ problem.**

**Aim:**

To write a python program to solve 8 Queens Problem

**Algorithm:**

**1.Start with an empty chessboard** (8x8 grid).

**2.Place a queen in the first column:**

Start by placing a queen in row 1 of column 1.

**3.Move to the next column:**

For the next column, try placing a queen in any row that is not under attack (i.e., no other queen in the same row, column, or diagonal).

**4.Check if the current placement is safe:**

If a queen can be safely placed in the current column, place it and move to the next column.

If no safe row is found in the current column, backtrack (i.e., remove the last placed queen from the previous column and try a different row).

**5.Repeat steps 3-4 until:**

All 8 queens are placed on the board, or Backtrack all the way to the first column if no solution is found.

**6.If all queens are placed:**

Print or store the solution.

**7.If no valid placement is found** after trying all rows and columns, declare that no solution exists.

file writer is a class

my writer is a object for that class

file writer class is available in java.io is a package

package means we should search for import only

include try and catch with exception

**program:**

def print\_board(board):  
 for row in board:  
 print(" ".join(row))  
 print("\n")  
  
def is\_safe(board, row, col):  
 n = len(board)  
  
 for i in range(col):  
 if board[row][i] == 'Q':  
 return False  
  
 for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  
 if board[i][j] == 'Q':  
 return False  
  
 for i, j in zip(range(row, n, 1), range(col, -1, -1)):  
 if board[i][j] == 'Q':  
 return False  
  
 return True  
  
def solve\_n\_queens(board, col):  
 n = len(board)  
  
 if col >= n:  
 print\_board(board)  
 return True  
  
 res = False  
 for i in range(n):  
 if is\_safe(board, i, col):  
 board[i][col] = 'Q'  
 res = solve\_n\_queens(board, col + 1) or res  
 board[i][col] = '.'  
  
 return res  
  
def solve\_8\_queens():  
 n = 8  
 board = [['.' for \_ in range(n)] for \_ in range(n)]  
  
 if not solve\_n\_queens(board, 0):  
 print("No solution exists")  
  
solve\_8\_queens()

**OUTPUT:**

Q . . . . . . .

. . . . . . Q .

. . . . Q . . .

. . . . . . . Q

. Q . . . . . .

. . . Q . . . .

. . . . . Q . .

. . Q . . . . .

**Result:**

The above python program has been executed successfully.

**2.Write a python program to solve water jug problem.**

**Aim:**

To solve water jug problem using Python program.

**Algorithm:**

**1.Start with two jugs** of different capacities: Jug X with capacity x liters, and Jug Y with capacity y liters.

**2.Goal:** Obtain exactly z liters of water in either of the jugs.

**3.Allowed operations:**

Fill any of the jugs completely. Empty any of the jugs.

Pour water from one jug to the other until one jug is empty or the other jug is full.

**4.Use BFS (Breadth-First Search)** to explore all possible states of water in the jugs:

Maintain a queue of states (amount\_in\_X, amount\_in\_Y). Perform all possible operations on the jugs for each state. Keep track of visited states to avoid cycles.

**5.Stop** when exactly z litters is achieved in either jug or both combined.

**Program:**

from collections import deque  
  
def is\_solvable(x, y, z):  
 if z > max(x, y) or z % gcd(x, y) != 0:  
 return False  
 return True  
  
def gcd(a, b):  
 while b != 0:  
 a, b = b, a % b  
 return a  
  
def water\_jug(x, y, z):  
 if not is\_solvable(x, y, z):  
 return "No solution"  
  
 visited = set()  
 queue = deque([(0, 0)])  
  
 while queue:  
 a, b = queue.popleft()  
  
 if a == z or b == z or a + b == z:  
 return f"Solution found: Jug X = {a}, Jug Y = {b}"  
  
 if (a, b) in visited:  
 continue  
  
 visited.add((a, b))  
  
 queue.append((x, b)) *# Fill jug X* queue.append((a, y)) *# Fill jug Y* queue.append((0, b)) *# Empty jug X* queue.append((a, 0)) *# Empty jug Y* queue.append((max(0, a - (y - b)), min(y, a + b))) *# Pour from X to Y* queue.append((min(x, a + b), max(0, b - (x - a)))) *# Pour from Y to X* return "No solution"  
  
x = 4  
y = 3  
z = 2  
  
print(water\_jug(x, y, z))

**OUTPUT:**

Solution found: Jug X = 4, Jug Y = 2

**Result:**

The above python program has been executed successfully.

**3.Write a python program to solve Crypt Arithmetic problem.**

**Aim:**

To solve Crypt Arithmetic problem using Python programming.

**Algorithm:**

**1.Cryptarithmetic Problem:**

Given an equation like SEND + MORE = MONEY, assign digits to letters such that the equation holds true.

**2.Objective:**

Each letter should represent a unique digit, and the resulting equation should be mathematically valid.

**3.Approach:**

Create all possible digit assignments for letters. Check each assignment to see if it satisfies the equation. Return the valid solution.

**Program:**

from itertools import permutations  
  
def is\_valid\_solution(s1, s2, s3, mapping):  
 num1 = int("".join(str(mapping[c]) for c in s1))  
 num2 = int("".join(str(mapping[c]) for c in s2))  
 num3 = int("".join(str(mapping[c]) for c in s3))  
 return num1 + num2 == num3  
  
def solve\_cryptarithmetic(s1, s2, s3):  
 letters = set(s1 + s2 + s3)  
 if len(letters) > 10:  
 return "No solution"  
  
 for perm in permutations(range(10), len(letters)):  
 mapping = dict(zip(letters, perm))  
 if is\_valid\_solution(s1, s2, s3, mapping):  
 return {letter: mapping[letter] for letter in letters}  
  
 return "No solution"  
  
s1 = "SEND"  
s2 = "MORE"  
s3 = "MONEY"  
  
print(solve\_cryptarithmetic(s1, s2, s3))

**OUTPUT:**

{'O': 0, 'R': 8, 'Y': 2, 'S': 9, 'D': 7, 'E': 5, 'M': 1, 'N': 6}

**Result:**

The above python program has been executed successfully.

**4.Write a Python program to implement BFS.**

**Aim:**

To implement Breadth First Search using python programming.

**Algorithm:**

**1.**Start at the source node.

**2.**Enqueue the source node to a queue.

**3.**Mark the source node as visited.

**4.**While the queue is not empty:

Dequeue a node from the front of the queue.

Process the dequeued node.

For each unvisited neighbor of the dequeued node:

Mark it as visited. Enqueue the neighbor.

**5.**Repeat until the queue is empty.

**Program:**

from collections import deque  
  
def bfs(graph, start):  
 visited = set()  
 queue = deque([start])  
 visited.add(start)  
  
 while queue:  
 node = queue.popleft()  
 print(node, end=" ")  
  
 for neighbor in graph[node]:  
 if neighbor not in visited:  
 visited.add(neighbor)  
 queue.append(neighbor)  
  
graph = {  
 'A': ['B', 'C', 'D'],  
 'B': ['E', 'F'],  
 'C': ['G'],  
 'D': ['H'],  
 'E': [],  
 'F': [],  
 'G': [],  
 'H': []  
}  
  
bfs(graph, 'A')

**OUTPUT:**

A B C D E F G H

**Result:**

The above python program has been executed successfully.

**5.Write a Python program to implement DFS.**

**Aim:**

To Implement Depth First Search using python programming.

**Algorithm:**

**1.**Start at the source node.

**2.**Mark the source node as visited.

**3.**Explore each neighbor of the current node:

For each unvisited neighbor, recursively apply DFS.

**4.**Repeat the process until all nodes are visited.

**Program:**

def dfs(graph, start, visited=None):  
 if visited is None:  
 visited = set()  
 visited.add(start)  
 print(start, end=" ")  
   
 for neighbor in graph[start]:  
 if neighbor not in visited:  
 dfs(graph, neighbor, visited)  
  
graph = {  
 'A': ['B', 'C', 'D'],  
 'B': ['E', 'F'],  
 'C': ['G'],  
 'D': ['H'],  
 'E': [],  
 'F': [],  
 'G': [],  
 'H': []  
}  
  
dfs(graph, 'A')

**OUTPUT:**

A B E F C G D H

**Result:**

The above python program has been executed successfully.

**6.Write a Python program to implement A\* Search.**

**Aim:**

To Implement A\* Search using python programming.

**Algorithm:**

**1.**Initialize the open list with the start node and set its cost g to 0.

**2.**Initialize the closed list as empty.

**3.**While the open list is not empty:

Find the node with the lowest f value (f = g + h) in the open list.

If the current node is the goal, return the path.

Move the current node from the open list to the closed list.

For each neighbor of the current node:

If the neighbor is already in the closed list, skip it.

If the neighbor is not in the open list or a shorter path is found:

Update its g and f values.

Set the current node as its parent.

Add it to the open list if not already there.

**4.**If the open list is empty and no path is found, return failure.

**Program:**

import heapq  
  
def a\_star\_search(graph, start, goal, h):  
 open\_list = []  
 heapq.heappush(open\_list, (0 + h[start], start))  
  
 g = {start: 0}  
 parents = {start: None}  
 closed\_list = set()  
  
 while open\_list:  
 current\_f, current\_node = heapq.heappop(open\_list)  
  
 if current\_node == goal:  
 path = []  
 while current\_node:  
 path.append(current\_node)  
 current\_node = parents[current\_node]  
 return path[::-1]  
  
 closed\_list.add(current\_node)  
  
 for neighbor, cost in graph[current\_node]:  
 if neighbor in closed\_list:  
 continue  
  
 tentative\_g = g[current\_node] + cost  
 if neighbor not in g or tentative\_g < g[neighbor]:  
 g[neighbor] = tentative\_g  
 f = tentative\_g + h[neighbor]  
 heapq.heappush(open\_list, (f, neighbor))  
 parents[neighbor] = current\_node  
  
 return None  
  
graph = {  
 'A': [('B', 1), ('C', 3)],  
 'B': [('D', 3), ('E', 1)],  
 'C': [('F', 5)],  
 'D': [('G', 1)],  
 'E': [('G', 4)],  
 'F': [('G', 2)],  
 'G': []  
}  
  
h = {  
 'A': 6,  
 'B': 4,  
 'C': 4,  
 'D': 2,  
 'E': 2,  
 'F': 2,  
 'G': 0  
}  
  
path = a\_star\_search(graph, 'A', 'G', h)  
print("Path:", path)

**OUTPUT:**

Path: ['A', 'B', 'D', 'G']\]]\\\]\]

**Result:**

The above python program has been executed successfully.

**7.Write a python program to implement map colouring for attaining CSP.**

**Aim:**

To Implement Map colouring for attaining CSP by using Python programming

**Algorithm:**

**1.**Start with an uncolored map.

**2.**Assign the first color to the first region.

**3.**Move to the next region:

Assign the smallest possible color that satisfies the constraint (no neighboring regions have the same color).

**4.**Backtrack if no valid color can be assigned to a region.

**5.**Repeat until all regions are colored.

**Program:**

def is\_safe(graph, colors, region, color):  
 for neighbor in graph[region]:  
 if colors[neighbor] == color:  
 return False  
 return True  
  
def map\_coloring(graph, m, colors, region):  
 if region == len(graph):  
 return True  
  
 for color in range(1, m + 1):  
 if is\_safe(graph, colors, region, color):  
 colors[region] = color  
 if map\_coloring(graph, m, colors, region + 1):  
 return True  
 colors[region] = 0  
 return False  
  
def solve\_map\_coloring(graph, m):  
 colors = [0] \* len(graph)  
 if map\_coloring(graph, m, colors, 0):  
 return colors  
 else:  
 return "No solution"  
  
graph = {  
 0: [1, 2],  
 1: [0, 2, 3],  
 2: [0, 1, 3],  
 3: [1, 2]  
}  
  
m = 3  
result = solve\_map\_coloring(graph, m)  
print("Solution:", result)

**OUTPUT:**

Solution: [1, 2, 3, 1]

**Result:**

The above python program has been executed successfully.

**8.Write a python program to implementing a TIC TAC TOE game.**

**Aim:**

To Implementing a TIC TAC TOE game by using Python programming.

**Algorithm:**

**1.**Initialize an empty 3x3 board.

**2.**Players take turns marking their symbol ('X' or 'O') on the board.

**3.**Check for a win after each move:

A player wins if they occupy an entire row, column, or diagonal.

**4.**Check for a tie if all positions are filled with no winner.

**5.**Repeat until either a player wins or the game ends in a tie.

**Program:**

def print\_board(board):  
 for row in board:  
 print(" | ".join(row))  
 print("-" \* 5)  
  
def check\_winner(board, player):  
 for row in board:  
 if all([s == player for s in row]):  
 return True  
 for col in range(3):  
 if all([board[row][col] == player for row in range(3)]):  
 return True  
 if all([board[i][i] == player for i in range(3)]) or all([board[i][2-i] == player for i in range(3)]):  
 return True  
 return False  
  
def check\_draw(board):  
 return all([cell != ' ' for row in board for cell in row])  
  
def tic\_tac\_toe():  
 board = [[' ' for \_ in range(3)] for \_ in range(3)]  
 current\_player = 'X'  
  
 while True:  
 print\_board(board)  
 row = int(input(f"Player {current\_player}, enter row (0, 1, 2): "))  
 col = int(input(f"Player {current\_player}, enter col (0, 1, 2): "))  
  
 if board[row][col] == ' ':  
 board[row][col] = current\_player  
 else:  
 print("Cell already taken, try again.")  
 continue  
  
 if check\_winner(board, current\_player):  
 print\_board(board)  
 print(f"Player {current\_player} wins!")  
 break  
  
 if check\_draw(board):  
 print\_board(board)  
 print("It's a draw!")  
 break  
  
 current\_player = 'O' if current\_player == 'X' else 'X'  
  
tic\_tac\_toe()

**OUTPUT:**

Player O, enter col (0, 1, 2): 1

| | X

-----

| |

-----

| O |

-----

Player X, enter row (0, 1, 2): 0

**Result:**

The above python program has been executed successfully.

**9.Write a python program to implement Travelling sales men program.**

**Aim:**

To Implement Travelling sales men program by using Python programming.

**Algorithm:**

**1.**Define a graph with n nodes and a distance matrix.

**2.**Use dynamic programming to explore all possible subsets of nodes.

**3.**For each subset, calculate the minimum cost to visit all nodes in the subset and return to the starting node.

**4.**Store intermediate results to avoid recalculating costs for the same subset.

**5.**Return the minimum cost for visiting all nodes and completing the tour.

**Program:**

import itertools  
  
def travelling\_salesman\_problem(graph, start):  
 n = len(graph)  
 all\_nodes = range(n)  
  
 dp = {}  
  
 *# Initialize dp table for single-node subsets excluding the start node* for subset\_size in range(1, n):  
 for subset in itertools.combinations(all\_nodes, subset\_size):  
 if start in subset:  
 continue  
 dp[(subset, subset[0])] = float('inf')  
  
 *# Set the starting point, no cost to start at the starting node* dp[((start,), start)] = 0  
  
 *# Fill dp table for larger subsets* for subset\_size in range(2, n + 1):  
 for subset in itertools.combinations(all\_nodes, subset\_size):  
 if start not in subset:  
 continue  
 for k in subset:  
 if k == start:  
 continue  
 prev\_subset = tuple([x for x in subset if x != k])  
 dp[(subset, k)] = float('inf')  
 for m in prev\_subset:  
 if (prev\_subset, m) in dp:  
 dp[(subset, k)] = min(dp[(subset, k)], dp[(prev\_subset, m)] + graph[m][k])  
  
 *# Calculate the minimum cost to complete the cycle* final\_res = min(dp[(tuple(all\_nodes), k)] + graph[k][start] for k in range(1, n))  
  
 return final\_res  
  
graph = [  
 [0, 10, 15, 20],  
 [10, 0, 35, 25],  
 [15, 35, 0, 30],  
 [20, 25, 30, 0]  
]  
  
start\_node = 0  
result = travelling\_salesman\_problem(graph, start\_node)  
print("Minimum cost:", result)

**OUTPUT:**

Minimum cost: 80

**Result:**

The above python program has been executed successfully.

**10.Write a Python program to implement alpha and beta tunning.**

**Aim:**

To implement alpha and beta pruning using Python programming.

**Algorithm:**

1.Start with initial alpha (worst case for maximizer, initially -∞) and beta (worst case for minimizer, initially ∞).

2.Traverse the tree:

If the current node is a maximizing player, update alpha (maximum value encountered).

If the current node is a minimizing player, update beta (minimum value encountered).

3.If alpha >= beta, prune the remaining branches (skip evaluating them).

4.Continue recursively until the leaves are reached.

**Program:**

import math  
  
*# Constants for infinity values*MAX, MIN = math.inf, -math.inf  
  
*# Alpha-Beta Pruning function*def alpha\_beta\_pruning(depth, nodeIndex, isMaximizing, values, alpha, beta):  
 *# Base case: when depth is maximum (reached leaf node)* if depth == 3:  
 return values[nodeIndex]  
  
 *# Maximizing player* if isMaximizing:  
 maxEval = MIN  
 *# Traverse two children* for i in range(2):  
 eval = alpha\_beta\_pruning(depth + 1, nodeIndex \* 2 + i, False, values, alpha, beta)  
 maxEval = max(maxEval, eval)  
 alpha = max(alpha, eval)  
 if beta <= alpha: *# Prune remaining branches* break  
 return maxEval  
  
 *# Minimizing player* else:  
 minEval = MAX  
 *# Traverse two children* for i in range(2):  
 eval = alpha\_beta\_pruning(depth + 1, nodeIndex \* 2 + i, True, values, alpha, beta)  
 minEval = min(minEval, eval)  
 beta = min(beta, eval)  
 if beta <= alpha: *# Prune remaining branches* break  
 return minEval  
  
*# Values at the leaf nodes of the game tree*values = [3, 5, 6, 9, 1, 2, 0, -1]  
  
*# Alpha and Beta initialization*alpha = MIN  
beta = MAX  
  
*# Function call to start the Alpha-Beta pruning process*optimal\_value = alpha\_beta\_pruning(0, 0, True, values, alpha, beta)  
  
*# Output the optimal value*print("The optimal value is:", optimal\_value)

**OUTPUT:**

The optimal value is: 5

**Result:**

The above python program has been executed successfully.

**11.Write a python program to implement decision tree.**

**Aim:**

To implement Decision tree using Python programming.

**Algorithm:**

1.Select the Best Attribute: The algorithm picks the best feature to split the data based on a splitting criterion (e.g., Gini Impurity or Information Gain).

2.Split the Dataset: Recursively split the dataset based on the best attribute.

3.Leaf Nodes: Once all data is perfectly classified, or a stopping criterion is met, the tree stops growing and assigns a class to the leaf nodes

**Program:**

*# Importing necessary libraries*from sklearn.datasets import load\_iris  
from sklearn.model\_selection import train\_test\_split  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix  
from sklearn import tree  
import matplotlib.pyplot as plt  
  
*# Load the iris dataset*iris = load\_iris()  
X = iris.data *# Features*y = iris.target *# Target labels  
  
# Split the dataset into training and testing sets*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)  
  
*# Initialize the Decision Tree classifier*clf = DecisionTreeClassifier()  
  
*# Train the model on the training data*clf.fit(X\_train, y\_train)  
  
*# Predict the labels for the test data*y\_pred = clf.predict(X\_test)  
  
*# Calculate accuracy*accuracy = accuracy\_score(y\_test, y\_pred)  
print(f'Accuracy: {accuracy\*100:.2f}%')  
  
*# Display classification report*print("\nClassification Report:")  
print(classification\_report(y\_test, y\_pred))  
  
*# Display confusion matrix*print("Confusion Matrix:")  
print(confusion\_matrix(y\_test, y\_pred))  
  
*# Plot the decision tree*plt.figure(figsize=(12, 8))  
tree.plot\_tree(clf, feature\_names=iris.feature\_names, class\_names=iris.target\_names, filled=True)  
plt.title("Decision Tree for Iris Dataset")  
plt.show()

**OUTPUT:**

Accuracy: 0.9777777777777777

Confusion Matrix:

[[14 0 0]

[ 0 13 1]

[ 0 0 15]]

Classification Report:

precision recall f1-score support

setosa 1.00 1.00 1.00 14

versicolor 1.00 0.93 0.96 14

virginica 0.94 1.00 0.97 15

accuracy 0.98 43

macro avg 0.98 0.98 0.98 43

weighted avg 0.98 0.98 0.98 43

**Result:**

The above python program has been executed successfully.

**12. Write a python program to implement feed forward neural network.**

**Aim:**

To Implement Feed Forward Neural Network using python programming

**Algorithm:**

1. Initialization:

Weights and Biases: Randomly initialize the weights and biases for each neuron in the network.

2. Forward Propagation:

Input Layer: Feed the input data to the input layer.

Hidden Layer:

Calculate the weighted sum of inputs and biases for each neuron.

Apply an activation function (e.g., sigmoid) to the result.

Output Layer:

Repeat the above steps for the hidden layer to get the final output.

3. Backpropagation:

Calculate Error: Compute the difference between the predicted output and the actual target.

Propagate Error: Propagate the error backward through the network, updating the weights and biases based on the error and the activation function's derivative.

4. Update Weights and Biases:

Adjust the weights and biases using gradient descent, taking a step in the direction that reduces the error.

5. Repeat:

Iterate through the training data multiple times (epochs), repeating steps 2-4 to refine the network's weights and biases.

6. Prediction:

Once the network is trained, feed new input data into the network using the same forward propagation steps to get the predicted output.

**Program:**

import numpy as np  
  
def sigmoid(x):  
 *"""  
 This function defines the sigmoid activation function,  
 commonly used in hidden layers of neural networks.  
 """* return 1 / (1 + np.exp(-x))  
  
def feedforward\_neural\_network(X, y, hidden\_size, learning\_rate, epochs):  
 *"""  
 This function trains a feedforward neural network.  
  
 Args:  
 X: Training input data (2D array)  
 y: Target output data (2D array)  
 hidden\_size: Number of neurons in the hidden layer  
 learning\_rate: Step size for weight updates during training  
 epochs: Number of training iterations  
  
 Returns:  
 W1, b1, W2, b2: Trained weights and biases for hidden and output layers  
 """  
  
 # Initialize weights and biases using Python's built-in random module* input\_size, output\_size = X.shape[1], y.shape[1]  
 W1 = np.random.rand(input\_size, hidden\_size)  
 b1 = np.zeros((1, hidden\_size))  
 W2 = np.random.rand(hidden\_size, output\_size)  
 b2 = np.zeros((1, output\_size))  
  
 *# Training loop* for epoch in range(epochs):  
 *# Forward propagation* Z1 = np.dot(X, W1) + b1 *# Weighted sum with bias for hidden layer* A1 = sigmoid(Z1) *# Apply sigmoid activation* Z2 = np.dot(A1, W2) + b2 *# Weighted sum with bias for output layer* y\_pred = sigmoid(Z2) *# Apply sigmoid activation (output prediction)  
  
 # Backpropagation* dZ2 = y\_pred - y *# Calculate error* dW2 = np.dot(A1.T, dZ2) *# Update weights for output layer using chain rule* db2 = np.sum(dZ2, axis=0, keepdims=True) *# Update biases for output layer* dZ1 = np.dot(dZ2, W2.T) \* (A1 \* (1 - A1)) *# Backpropagate error* dW1 = np.dot(X.T, dZ1) *# Update weights for hidden layer* db1 = np.sum(dZ1, axis=0, keepdims=True) *# Update biases for hidden layer  
  
 # Update weights and biases with learning rate* W1 -= learning\_rate \* dW1  
 b1 -= learning\_rate \* db1  
 W2 -= learning\_rate \* dW2  
 b2 -= learning\_rate \* db2  
  
 return W1, b1, W2, b2  
  
*# Example usage (replace with your own data)*X = np.array([[0, 1], [1, 0], [1, 1], [0, 0]]) *# Input data*y = np.array([[0], [1], [1], [0]]) *# Target output*hidden\_size = 4  
learning\_rate = 0.1  
epochs = 1000  
  
W1, b1, W2, b2 = feedforward\_neural\_network(X, y, hidden\_size, learning\_rate, epochs)  
  
*# Use the trained network for predictions (replace with new input)*new\_input = np.array([[0, 1]])  
Z1 = np.dot(new\_input, W1) + b1  
A1 = sigmoid(Z1)  
Z2 = np.dot(A1, W2) + b2  
y\_pred = sigmoid(Z2)  
print(y\_pred)

**OUTPUT:**

[[0.00694746]]

**Result:**

The above python program has been executed successfully.

**13.Write the python program for solving 8 puzzle problem.**

**Aim:**

To 8 puzzle problem by using python program.

**Algorithm:**

1.Initialize:

Set the start state and goal state.

Calculate the Manhattan Distance for the start state.

2.Create Open and Closed Lists:

Open list: contains states to explore.

Closed set: tracks explored states.

3.Search Loop:

While the open list is not empty:

Remove the state with the lowest cost f=g+h.

If this state is the goal, stop and return the solution. Generate neighboring states.

If the neighbor hasn't been explored, add it to the open list.

4.Solution:

When the goal state is reached, backtrack to get the sequence of moves

**Program:**

import heapq  
  
class PuzzleState:  
 def \_\_init\_\_(self, board, parent=None, move="", g=0, h=0):  
 self.board = board  
 self.parent = parent  
 self.move = move  
 self.g = g  
 self.h = h  
 self.f = g + h  
  
 def \_\_lt\_\_(self, other):  
 return self.f < other.f  
  
def manhattan\_distance(board):  
 distance = 0  
 goal = {(0, 0): 1, (0, 1): 2, (0, 2): 3,  
 (1, 0): 4, (1, 1): 5, (1, 2): 6,  
 (2, 0): 7, (2, 1): 8, (2, 2): 0}  
 for i in range(3):  
 for j in range(3):  
 value = board[i][j]  
 if value != 0:  
 goal\_x, goal\_y = [(k, l) for (k, l), v in goal.items() if v == value][0]  
 distance += abs(goal\_x - i) + abs(goal\_y - j)  
 return distance  
  
def get\_neighbors(board):  
 neighbors = []  
 empty = [(i, j) for i in range(3) for j in range(3) if board[i][j] == 0][0]  
 i, j = empty  
 directions = [('Up', (i - 1, j)), ('Down', (i + 1, j)), ('Left', (i, j - 1)), ('Right', (i, j + 1))]  
 for direction, (x, y) in directions:  
 if 0 <= x < 3 and 0 <= y < 3:  
 new\_board = [row[:] for row in board]  
 new\_board[i][j], new\_board[x][y] = new\_board[x][y], new\_board[i][j]  
 neighbors.append((direction, new\_board))  
 return neighbors  
  
def astar(start):  
 open\_list = []  
 closed\_set = set()  
 start\_state = PuzzleState(start, g=0, h=manhattan\_distance(start))  
 heapq.heappush(open\_list, start\_state)  
  
 while open\_list:  
 current\_state = heapq.heappop(open\_list)  
  
 if current\_state.h == 0:  
 solution = []  
 while current\_state.parent:  
 solution.append(current\_state.move)  
 current\_state = current\_state.parent  
 return solution[::-1]  
  
 closed\_set.add(tuple(map(tuple, current\_state.board)))  
  
 for move, neighbor in get\_neighbors(current\_state.board):  
 neighbor\_state = PuzzleState(neighbor, parent=current\_state, move=move,  
 g=current\_state.g + 1, h=manhattan\_distance(neighbor))  
  
 if tuple(map(tuple, neighbor)) not in closed\_set:  
 heapq.heappush(open\_list, neighbor\_state)  
  
 return None  
  
start\_board = [[2, 8, 3], [1, 6, 4], [7, 0, 5]]  
solution = astar(start\_board)  
print("Solution moves:", solution)

**OUTPUT:**

Solution moves: None

**Result:**

The above python program has been executed successfully.

**14.Write the python program to implement Vacuum cleaner problem.**

**Aim:**

To implement Vacuum cleaner problem using Python Program.

**Algorithm:**

1.Initialize the Environment:

Set up the environment (grid) with cells either clean or dirty.

Place the vacuum cleaner in a random or specific position in the environment.

Perceive the Environment:

2.Check the current location: if it's dirty, clean it.

Action:

If the current cell is dirty, clean it.

If the cell is clean, move the vacuum to a neighboring cell (left, right, up, down).

3.Repeat:

Continue cleaning and moving until all cells are clean.

4.Stop:

End when there are no more dirty cells.

**Program:**

import random  
  
class VacuumCleaner:  
 def \_\_init\_\_(self, environment, start):  
 self.environment = environment  
 self.position = start  
 self.cleaned = 0  
  
 def clean(self):  
 if self.environment[self.position[0]][self.position[1]] == 1:  
 self.environment[self.position[0]][self.position[1]] = 0  
 self.cleaned += 1  
  
 def move(self):  
 directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  
 random.shuffle(directions)  
 for d in directions:  
 new\_position = (self.position[0] + d[0], self.position[1] + d[1])  
 if 0 <= new\_position[0] < len(self.environment) and 0 <= new\_position[1] < len(self.environment[0]):  
 self.position = new\_position  
 break  
  
def is\_clean(environment):  
 return all(all(cell == 0 for cell in row) for row in environment)  
  
environment = [  
 [1, 0, 1],  
 [1, 1, 0],  
 [0, 1, 1]  
]  
  
vacuum = VacuumCleaner(environment, (0, 0))  
  
while not is\_clean(environment):  
 vacuum.clean()  
 vacuum.move()  
  
print("All cells are clean!")  
print("Cleaned cells:", vacuum.cleaned)

**OUTPUT:**

All cells are clean!

Cleaned cells: 6

**Result:**

The above python program has been executed successfully.

**15. Write the python program Missionaries cannibal problem.**

**Aim:**

To Solve Missionaries Cannibal Problem using python Program.

**Algorithm:**

1.Initialize the Problem State:

Define the initial state: 3 missionaries, 3 cannibals, and a boat on the left bank.

The goal state: all missionaries and cannibals are safely on the right bank.

2.Define Valid Moves:

Define possible moves (1 or 2 people can cross in the boat).

Ensure the moves do not violate safety conditions (missionaries cannot be outnumbered by cannibals on either bank).

3.Breadth-First Search (BFS):

Use BFS to explore all possible states.

Start from the initial state.

For each state, generate valid successor states (valid moves).

Track visited states to avoid revisiting them.

4.Goal Check:

If a generated state is the goal state, return the solution (sequence of moves).

5.Repeat:

Continue until the goal is reached or all possible states are explored.

**Program:**

from collections import deque  
  
def is\_valid(state):  
 m\_left, c\_left, boat, m\_right, c\_right = state  
 if m\_left < 0 or m\_right < 0 or c\_left < 0 or c\_right < 0:  
 return False  
 if (m\_left > 0 and m\_left < c\_left) or (m\_right > 0 and m\_right < c\_right):  
 return False  
 return True  
  
def get\_successors(state):  
 successors = []  
 m\_left, c\_left, boat, m\_right, c\_right = state  
 moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]  
 for m, c in moves:  
 if boat == 1:  
 new\_state = (m\_left - m, c\_left - c, 0, m\_right + m, c\_right + c)  
 else:  
 new\_state = (m\_left + m, c\_left + c, 1, m\_right - m, c\_right - c)  
 if is\_valid(new\_state):  
 successors.append(new\_state)  
 return successors  
  
def bfs(start\_state, goal\_state):  
 queue = deque([(start\_state, [])])  
 visited = set()  
 visited.add(start\_state)  
  
 while queue:  
 current\_state, path = queue.popleft()  
 if current\_state == goal\_state:  
 return path  
  
 for successor in get\_successors(current\_state):  
 if successor not in visited:  
 visited.add(successor)  
 queue.append((successor, path + [successor]))  
  
start\_state = (3, 3, 1, 0, 0)  
goal\_state = (0, 0, 0, 3, 3)  
  
solution = bfs(start\_state, goal\_state)  
print("Solution:", solution)

**OUTPUT:**

Solution: [(3, 1, 0, 0, 2), (3, 2, 1, 0, 1), (3, 0, 0, 0, 3), (3, 1, 1, 0, 2), (1, 1, 0, 2, 2), (2, 2, 1, 1, 1), (0, 2, 0, 3, 1), (0, 3, 1, 3, 0), (0, 1, 0, 3, 2), (1, 1, 1, 2, 2), (0, 0, 0, 3, 3)]

**Result:**

The above python program has been executed successfully.

**PROLOG PROGRAMS**

**1.Write a Prolog program to sum the integers from 1 to n.**

**Aim:**

To Write a prolog program for Sum the integers from 1 to n.

**Algorithm:**

 **Input:** An integer NNN.

 **Output:** The sum of integers from 1 to NNN.

 **Base Case:**

* If N=0N = 0N=0, return 0 (the sum of numbers up to 0 is 0).

 **Recursive Case:**

* If N>0N > 0N>0:
  1. Subtract 1 from NNN to get N−1N - 1N−1.
  2. Recursively calculate the sum of integers from 1 to N−1N-1N−1 and store the result.
  3. Add NNN to the result from the recursive step.

 **Return the result** (sum of NNN plus the sum from the recursive call).

**Program:**

% Base case: sum of numbers from 1 to 0 is 0.

sum\_to\_n(0, 0).

% Recursive case: sum of numbers from 1 to N is N + sum of numbers from 1 to N-1.

sum\_to\_n(N, Sum) :-

N > 0,

N1 is N - 1,

sum\_to\_n(N1, Sum1),

Sum is N + Sum1.

**OUTPUT:**

?- sum\_to\_n(5, Sum).

Sum = 15.

**Result:**

The above Prolog program has been executed successfully.

**2.Write a Prolog Program for A DB with NAME, DOB.**

**Aim:**

To Write a Prolog Program for A DB with NAME, DOB.

**Algorithm:**

1. **Input:**
   * A set of facts containing the **Name**, **Day**, **Month**, and **Year** of birth for several people.
2. **Output:**
   * The results of queries on the database such as finding the DOB of a specific person or listing people born in a specific month or year.
3. **Steps:**
   * **Database Setup:**
     + Define the facts (dob/4) for each person with their **Name**, **Day**, **Month**, and **Year**.
   * **Queries:**
     + **Find DOB:**
       - When a query like find\_dob(Name, Day, Month, Year) is called, it searches the dob/4 facts to match the Name and retrieve the corresponding **Day**, **Month**, and **Year**.
     + **Find People by Month:**
       - For a query like born\_in\_month(Month, Name), the program searches the dob/4 facts where the **Month** matches the input and returns the corresponding Name.
     + **Find People by Year:**
       - For a query like born\_in\_year(Year, Name), the program searches the dob/4 facts where the **Year** matches the input and returns the corresponding Name.
4. **Termination:**
   * The query returns all matching facts or terminates if there are no more results.

**Program:**

% Facts: name and date of birth (dob(Name, Day, Month, Year))

dob(john, 12, may, 1990).

dob(mary, 23, june, 1985).

dob(alex, 4, january, 2000).

dob(lisa, 15, september, 1992).

dob(peter, 10, march, 1988).

% Rule to find a person's name and DOB

find\_dob(Name, Day, Month, Year) :-

dob(Name, Day, Month, Year).

% Rule to find all people born in a specific month

born\_in\_month(Month, Name) :-

dob(Name, \_, Month, \_).

% Rule to find all people born in a specific year

born\_in\_year(Year, Name) :-

dob(Name, \_, \_, Year).

**OUTPUT:**

?- find\_dob(john, Day, Month, Year).

Day = 12,

Month = may,

Year = 1990.

**OP – 2**

?- born\_in\_month(may, Name).

Name = john.

**Result:**

The above Prolog program has been executed successfully.

**3.Write a Prolog program for STUDENT – TEACHER – SUB – CODE.**

**Aim:**

To Write a Prolog program for STUDENT – TEACHER – SUB – CODE

**Algorithm:**

 **Input:**

* A set of facts representing the relationship between **Student**, **Teacher**, **Subject**, and **Code**.

 **Output:**

* Answers to queries about students, teachers, subjects, and subject codes.

 **Steps:**

1. **Database Setup:**
   * Define the facts (student\_teacher\_subject\_code/4) for each **Student**, **Teacher**, **Subject**, and **Code**.
2. **Queries:**
   * **Find Subject and Code for Student:**
     + When find\_subject\_and\_code(Student, Subject, Code) is queried, it searches the student\_teacher\_subject\_code/4 facts for the corresponding **Student** and retrieves the **Subject** and **Code**.
   * **Find Teacher for Student:**
     + When find\_teacher(Student, Teacher) is queried, it searches the student\_teacher\_subject\_code/4 facts and returns the **Teacher** for the specified **Student**.
   * **Find Students of a Teacher:**
     + When students\_of\_teacher(Teacher, Student) is queried, it searches the facts and returns the **Student** for the specified **Teacher**.
   * **Find Students in Subject:**
     + When students\_in\_subject(Subject, Student) is queried, it returns all **Students** enrolled in the specified **Subject**.
   * **Find Subject Code by Teacher:**
     + When find\_subject\_code\_by\_teacher(Teacher, Subject, Code) is queried, it returns the **Subject** and **Code** taught by the specified **Teacher**.

 **Termination:**

* The query returns all matching facts or terminates when there are no more results.

**Program:**

% Facts: student\_teacher\_subject\_code(Student, Teacher, Subject, Code)

student\_teacher\_subject\_code(john, smith, mathematics, 'M101').

student\_teacher\_subject\_code(mary, smith, mathematics, 'M101').

student\_teacher\_subject\_code(alex, brown, physics, 'P201').

student\_teacher\_subject\_code(lisa, williams, chemistry, 'C301').

student\_teacher\_subject\_code(peter, taylor, biology, 'B401').

student\_teacher\_subject\_code(jane, taylor, biology, 'B401').

% Rule to find the subject and code a student is enrolled in

find\_subject\_and\_code(Student, Subject, Code) :-

student\_teacher\_subject\_code(Student, \_, Subject, Code).

% Rule to find the teacher of a specific student

find\_teacher(Student, Teacher) :-

student\_teacher\_subject\_code(Student, Teacher, \_, \_).

% Rule to find all students taught by a specific teacher

students\_of\_teacher(Teacher, Student) :-

student\_teacher\_subject\_code(Student, Teacher, \_, \_).

% Rule to find students enrolled in a specific subject

students\_in\_subject(Subject, Student) :-

student\_teacher\_subject\_code(Student, \_, Subject, \_).

% Rule to find the code of a subject taught by a teacher

find\_subject\_code\_by\_teacher(Teacher, Subject, Code) :-

student\_teacher\_subject\_code(\_, Teacher, Subject, Code).

**OUTPUT:**

?- find\_subject\_and\_code(john, Subject, Code).

Subject = mathematics,

Code = 'M101'.

**O/P – 2**

?- find\_teacher(jane, Teacher).

Teacher = taylor.

**Result:**

The above Prolog program has been executed successfully.

**4.Write a Prolog program for PLANATES DB.**

**Aim:**

To Write a Prolog program for PLANATES DB.

**Algorithm:**

 **Input:**

* A set of facts representing the planets with attributes: **Name**, **Distance from the Sun**, **Diameter**, and **Number of Moons**.

 **Output:**

* The answers to queries about planets based on their attributes.

 **Steps:**

1. **Database Setup:**
   * Define the facts using the predicate planet/4, which stores data about each planet, including **Name**, **Distance from the Sun**, **Diameter**, and **Number of Moons**.
2. **Queries:**
   * **Find Planet Details:**
     + When find\_planet\_details(Name, Distance, Diameter, Moons) is queried, it searches the planet/4 facts and returns the corresponding details for the planet with the matching name.
   * **Find Planets with Moons:**
     + When planets\_with\_moons(Moons, Name) is queried, it checks the planet/4 facts to return all planets that have more than the specified number of moons.
   * **Find Planets within Distance:**
     + When planets\_within\_distance(MaxDistance, Name) is queried, it returns the planets whose **Distance from the Sun** is less than or equal to the specified maximum distance.
   * **Find Planets Larger than Diameter:**
     + When planets\_larger\_than(Diameter, Name) is queried, it returns all planets with a diameter larger than the specified value.

 **Termination:**

* The query returns all matching planets or terminates if no more matches are found.

**Program:**

% Facts: planet(Name, DistanceFromSun (in million km), Diameter (in km), NumberOfMoons)

planet(mercury, 57.9, 4879, 0).

planet(venus, 108.2, 12104, 0).

planet(earth, 149.6, 12742, 1).

planet(mars, 227.9, 6779, 2).

planet(jupiter, 778.3, 139820, 79).

planet(saturn, 1427.0, 116460, 83).

planet(uranus, 2871.0, 50724, 27).

planet(neptune, 4495.1, 49244, 14).

% Rule to find the details of a specific planet

find\_planet\_details(Name, Distance, Diameter, Moons) :-

planet(Name, Distance, Diameter, Moons).

% Rule to find planets with more than a given number of moons

planets\_with\_moons(Moons, Name) :-

planet(Name, \_, \_, NumberOfMoons),

NumberOfMoons > Moons.

% Rule to find the planets within a specific distance from the sun

planets\_within\_distance(MaxDistance, Name) :-

planet(Name, Distance, \_, \_),

Distance =< MaxDistance.

% Rule to find planets larger than a certain diameter

planets\_larger\_than(Diameter, Name) :-

planet(Name, \_, PlanetDiameter, \_),

PlanetDiameter > Diameter.

**OUTPUT:**

?- find\_planet\_details(earth, Distance, Diameter, Moons).

Distance = 149.6,

Diameter = 12742,

Moons = 1.

**Result:**

The above Prolog program has been executed successfully.

**5.Write a Prolog program to implement Tower of Hanoi.**

**Aim:**

To Write a Prolog program to implement Tower of Hanoi.

**Algorithm:**

 **Input:**

* The number of disks **N**.
* The names of the three rods: **Source**, **Destination**, and **Auxiliary**.

 **Output:**

* The sequence of steps to move all disks from the **Source** rod to the **Destination** rod.

 **Steps:**

1. **Base Case:** If there is only 1 disk:
   * Move it directly from the **Source** rod to the **Destination** rod.
2. **Recursive Case:** For more than 1 disk:
   * Move the top **N-1** disks from **Source** to **Auxiliary** (using **Destination** as auxiliary).
   * Move the largest disk (disk **N**) from **Source** to **Destination**.
   * Move the **N-1** disks from **Auxiliary** to **Destination** (using **Source** as auxiliary).

 **Termination:**

* The recursion terminates when **N = 1**, at which point the base case is executed.

**Program:**

% Base case: Moving 1 disk from Source to Destination

hanoi(1, Source, Destination, \_) :-

write('Move disk 1 from '),

write(Source),

write(' to '),

write(Destination),

nl.

% Recursive case: Move N disks from Source to Destination using Auxiliary

hanoi(N, Source, Destination, Auxiliary) :-

N > 1,

N1 is N - 1,

% Move N-1 disks from Source to Auxiliary using Destination as Auxiliary

hanoi(N1, Source, Auxiliary, Destination),

% Move the Nth disk from Source to Destination

write('Move disk '),

write(N),

write(' from '),

write(Source),

write(' to '),

write(Destination),

nl,

% Move N-1 disks from Auxiliary to Destination using Source as Auxiliary

hanoi(N1, Auxiliary, Destination, Source).

**OUTPUT:**

?- hanoi(3, 'A', 'C', 'B').

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

**Result:**

The above Prolog program has been executed successfully.

**6.Write a Prolog program to print particular bird can fly or not. Incorporate required queries.**

**Aim:**

To Write a Prolog program to print particular bird can fly or not. Incorporate required queries.

**Algorithm:**

 **Input:**

* The name of the bird that you want to check if it can fly or not.

 **Output:**

* A statement indicating whether the bird can fly or not.

 **Steps:**

1. **Check if the Bird Exists:**
   * Use the bird/1 predicate to verify that the bird is in the database.
2. **Check if the Bird is Flightless:**
   * If the bird exists and is listed in the flightless/1 predicate, it cannot fly. The second rule of can\_fly/1 applies.
3. **Check if the Bird Can Fly:**
   * If the bird exists but is **not flightless** (\+ flightless(Bird)), it can fly. The first rule of can\_fly/1 applies.

 **Termination:**

* The program will output the result and stop once the correct rule is matched.

**Program:**

% Facts: bird(Bird) defines that the given Bird is a type of bird.

bird(sparrow).

bird(penguin).

bird(ostrich).

bird(eagle).

bird(parrot).

% Facts: flightless(Bird) defines birds that cannot fly.

flightless(penguin).

flightless(ostrich).

% Rule to determine if a bird can fly

can\_fly(Bird) :-

bird(Bird),

\+ flightless(Bird),

write(Bird), write(' can fly.'), nl.

% Rule to determine if a bird cannot fly

can\_fly(Bird) :-

bird(Bird),

flightless(Bird),

write(Bird), write(' cannot fly.'), nl.

**OUTPUT:**

?- can\_fly(sparrow).

sparrow can fly.

**O/P – 2**

?- can\_fly(penguin).

penguin cannot fly.

**Result:**

The above Prolog program has been executed successfully.

**7.Write a Prolog program to implement family tree.**

**Aim:**

To Write a Prolog program to implement family tree**.**

**Algorithm:**

**1. Define Family Members (Facts)**

* Step 1: Start by defining the parent relationships in the family tree. This includes specifying who is a parent of whom.
  + Example: parent(john, mary). means John is the parent of Mary.
* Step 2: Optionally, define the gender of each family member using facts male(X) and female(X) for clarity and more specific relationships.
  + Example: male(john)., female(mary).

**2. Define Basic Relationships (Rules)**

* Step 3: Define the **father** relationship as a rule.
  + Rule: father(X, Y) is true if X is a parent(X, Y) and male(X).
* Step 4: Define the **mother** relationship as a rule.
  + Rule: mother(X, Y) is true if X is a parent(X, Y) and female(X).
* Step 5: Define the **sibling** relationship.
  + Rule: sibling(X, Y) is true if both share at least one parent and are not the same person.

**3. Define Advanced Relationships (Rules)**

* Step 6: Define the **grandparent** relationship.
  + Rule: grandparent(X, Y) is true if X is a parent of Z and Z is a parent of Y.
* Step 7: Define **ancestor** recursively.
  + Rule: ancestor(X, Y) is true if X is a parent of Y or X is a parent of someone who is an ancestor of Y.
* Step 8: Define **uncle** and **aunt** relationships.
  + Rule: uncle(X, Y) is true if X is a male sibling of Y's parent.
  + Rule: aunt(X, Y) is true if X is a female sibling of Y's parent.
* Step 9: Define **cousin** relationship.
  + Rule: cousin(X, Y) is true if X's parent is a sibling of Y's parent.

**4. Query the Family Tree**

* Step 10: To retrieve relationships or check if a relationship exists, you can run **queries** in Prolog:

**Program:**

% Facts: Define the family members and their relationships

% Parent relationships

parent(john, mary). % John is a parent of Mary

parent(john, james). % John is a parent of James

parent(susan, mary). % Susan is a parent of Mary

parent(susan, james). % Susan is a parent of James

parent(mary, alice). % Mary is a parent of Alice

parent(mary, bob). % Mary is a parent of Bob

parent(peter, alice). % Peter is a parent of Alice

parent(peter, bob). % Peter is a parent of Bob

% Gender (optional but useful for some relationships)

male(john).

male(james).

male(peter).

male(bob).

female(susan).

female(mary).

female(alice).

% Rules: Define relationships

% Father: X is the father of Y if X is a parent of Y and X is male

father(X, Y) :-

parent(X, Y),

male(X).

% Mother: X is the mother of Y if X is a parent of Y and X is female

mother(X, Y) :-

parent(X, Y),

female(X).

% Sibling: X and Y are siblings if they share at least one parent

sibling(X, Y) :-

parent(Z, X),

parent(Z, Y),

X \= Y.

% Grandparent: X is a grandparent of Y if X is a parent of Z and Z is a parent of Y

grandparent(X, Y) :-

parent(X, Z),

parent(Z, Y).

% Grandfather: X is the grandfather of Y if X is a grandparent and male

grandfather(X, Y) :-

grandparent(X, Y),

male(X).

% Grandmother: X is the grandmother of Y if X is a grandparent and female

grandmother(X, Y) :-

grandparent(X, Y),

female(X).

% Ancestor: X is an ancestor of Y if X is a parent of Y or X is a parent of Z and Z is an ancestor of Y

ancestor(X, Y) :-

parent(X, Y).

ancestor(X, Y) :-

parent(X, Z),

ancestor(Z, Y).

% Uncle: X is the uncle of Y if X is male and the sibling of Y's parent

uncle(X, Y) :-

male(X),

sibling(X, Z),

parent(Z, Y).

% Aunt: X is the aunt of Y if X is female and the sibling of Y's parent

aunt(X, Y) :-

female(X),

sibling(X, Z),

parent(Z, Y).

% Cousin: X and Y are cousins if X's parent is the sibling of Y's parent

cousin(X, Y) :-

parent(A, X),

parent(B, Y),

sibling(A, B).

% Example Queries:

% ?- father(john, mary). % Check if John is the father of Mary

% ?- mother(mary, bob). % Check if Mary is the mother of Bob

% ?- grandparent(john, alice). % Check if John is the grandparent of Alice

% ?- ancestor(john, alice). % Check if John is an ancestor of Alice

% ?- uncle(james, alice). % Check if James is the uncle of Alice

% ?- cousin(bob, james). % Check if Bob and James are cousins

**OUTPUT:**

?- father(X, mary).

?- ancestor(X, bob).

**Result:**

The above Prolog program has been executed successfully.

**8.Write a Prolog program to Suggest Dieting System based on Disease.**

**Aim:**

To Write a Prolog program to Suggest Dieting System based on Disease.

**Algorithm:**

**Program:**

% Facts: Disease and Recommended Diet

% Diabetes diet recommendations

diet(diabetes, "Low carbohydrate, high fiber, moderate protein diet, avoid sugar").

% Hypertension diet recommendations

diet(hypertension, "Low sodium, rich in potassium and magnesium, DASH diet").

% Heart disease diet recommendations

diet(heart\_disease, "Low saturated fat, high in omega-3 fatty acids, lots of fruits and vegetables").

% Obesity diet recommendations

diet(obesity, "Calorie-controlled, high fiber, low-fat diet with plenty of fruits and vegetables").

% High cholesterol diet recommendations

diet(high\_cholesterol, "Low saturated fat, rich in soluble fiber, healthy fats like omega-3").

% Anemia diet recommendations

diet(anemia, "Iron-rich foods, vitamin B12, folic acid, and vitamin C to increase iron absorption").

% Constipation diet recommendations

diet(constipation, "High fiber, plenty of fluids, avoid processed foods").

% Gastroesophageal reflux disease (GERD) diet recommendations

diet(gerd, "Low-fat, avoid spicy foods, avoid citrus, eat small frequent meals").

% Kidney disease diet recommendations

diet(kidney\_disease, "Low sodium, potassium, and phosphorus, control protein intake").

% Cancer diet recommendations

diet(cancer, "High in fruits, vegetables, lean protein, healthy fats, and whole grains").

% Rules: Suggest Diet based on Disease

suggest\_diet(Disease, Diet) :-

diet(Disease, Diet).

% Example Queries:

% ?- suggest\_diet(diabetes, Diet). % Suggest diet for diabetes

% ?- suggest\_diet(hypertension, Diet). % Suggest diet for hypertension

% ?- suggest\_diet(anemia, Diet). % Suggest diet for anemia

**OUTPUT:**

?- suggest\_diet(hypertension, Diet).

Diet = "Low sodium, rich in potassium and magnesium, DASH diet".

**O/P:**

?- suggest\_diet(obesity, Diet).

Diet = "Calorie-controlled, high fiber, low-fat diet with plenty of fruits and vegetables".

**Result:**

The above Prolog program has been executed successfully.

**9.Write a Prolog Program to Implement Monkey Banana Problem.**

**Aim:**

To Write a Prolog Program to Implement Monkey Banana Problem.

**Algorithm:**

1. **Define the Initial State**

* Represent the initial state of the environment:
  + Monkey is on the ground.
  + The box is on the ground.
  + The bananas are hanging.

#### 2. ****Define the Goal State****

* The goal state is reached when the bananas are taken by the monkey.

#### 3. ****Define Possible Actions****

* Define the actions the monkey can perform:
  + **Move**: Move the monkey to a different position (e.g., from ground to tree).
  + **Climb**: Climb onto the box if on the ground.
  + **Push**: Push the box to a new position (e.g., under the bananas).
  + **Take**: Take the bananas if the monkey is on the box.

#### 4. ****Define Movement Capabilities****

* Specify which positions the monkey can move to and where the box can be pushed:
  + Monkey can move from:
    - Ground ↔ Tree
    - Ground ↔ Box
  + Box can move from:
    - Ground ↔ Under Banana

#### 5. ****Plan to Reach the Goal****

* Implement a planning function that:
  + Checks if the current state is the goal state.
  + If not, iterates through possible actions to generate a new state.
  + Recursively calls the planning function with the new state until the goal is reached.

#### 6. ****Backtrack if Necessary****

* If a series of actions does not lead to the goal state, backtrack to try alternative actions.

**Program:**

% Initial state:

% monkey is on the ground, box is on the ground, bananas are hanging.

state(ground, ground, hanging). % state(Monkey Position, Box Position, Banana Status)

% Possible actions

% Move the monkey to a different position

action(move(MonkeyNewPosition), state(MonkeyCurrentPosition, BoxPosition, BananaStatus), state(MonkeyNewPosition, BoxPosition, BananaStatus)) :-

monkey\_can\_move(MonkeyCurrentPosition, MonkeyNewPosition).

% Climb the box

action(climb, state(ground, on\_box, BananaStatus), state(on\_box, on\_box, BananaStatus)).

% Push the box

action(push, state(MonkeyPosition, BoxCurrentPosition, BananaStatus), state(MonkeyPosition, BoxNewPosition, BananaStatus)) :-

box\_can\_move(BoxCurrentPosition, BoxNewPosition).

% Take the bananas if the monkey is on the box

action(take, state(on\_box, BoxPosition, hanging), state(on\_box, BoxPosition, taken)).

% Define movement capabilities

monkey\_can\_move(ground, tree).

monkey\_can\_move(tree, ground).

monkey\_can\_move(ground, box).

% Define box movement capabilities

box\_can\_move(ground, under\_banana).

box\_can\_move(under\_banana, ground).

% Goal state: The bananas are taken

goal(state(\_, \_, taken)).

% Plan to reach the goal state

plan(State, []) :- goal(State).

plan(State, [Action|Rest]) :-

action(Action, State, NewState),

plan(NewState, Rest).

% Example Query: Find a plan to get the bananas

% ?- plan(state(ground, ground, hanging), Actions).

**OUTPUT:**

?- plan(state(ground, ground, hanging), Actions).

Actions = [move(box), push, climb, take].

**Result:**

The above Prolog program has been executed successfully.

**10.Write a Prolog program for fruits and its colour using back tracking.**

**Aim:**

To Write a Prolog program for fruits and its colour using back tracking.

**Algorithm:**

1. **Define Facts**: Represent fruits and their colors as facts.
2. **Define Rules**: Create rules to query fruits based on color and colors based on fruits.
3. **Backtracking Mechanism**: Allow Prolog's built-in backtracking to find all possible matches.
4. **User Queries**: Implement queries to retrieve information about fruits and their colors.

**Program:**

% Facts: Define fruits and their colors

fruit\_color(apple, red).

fruit\_color(banana, yellow).

fruit\_color(grape, purple).

fruit\_color(orange, orange).

fruit\_color(lemon, yellow).

fruit\_color(watermelon, green).

fruit\_color(blueberry, blue).

fruit\_color(strawberry, red).

% Rules: Define queries for fruits based on color and colors based on fruits

find\_fruit\_by\_color(Color, Fruit) :-

fruit\_color(Fruit, Color).

find\_color\_by\_fruit(Fruit, Color) :-

fruit\_color(Fruit, Color).

% Example Queries:

% ?- find\_fruit\_by\_color(yellow, Fruit).

% ?- find\_color\_by\_fruit(apple, Color).

**OUTPUT:**

?- find\_fruit\_by\_color(yellow, Fruit).

Fruit = banana ;

Fruit = lemon.

**Result:**

The above Prolog program has been executed successfully.

**11.Write a Prolog Program to Implement Best First Search Algorithm.**

**Aim:**

To Write a Prolog Program to Implement Best First Search Algorithm**.**

**Algorithm:**

### Algorithm: Best First Search

#### 1. ****Define the Graph****

* Represent the graph using edges and their associated costs.

#### 2. ****Define the Heuristic Function****

* Create a heuristic function that estimates the cost to reach the goal from any given node.

#### 3. ****Initialize the Search****

* Start with an initial path containing the starting node.

#### 4. ****Expand the Current Node****

* While there are paths to explore:
  1. **Select the Best Path**:
     + Choose the path with the lowest estimated total cost (current path cost + heuristic).
  2. **Check Goal**:
     + If the current node is the goal, return the path and its cost.
  3. **Generate Successors**:
     + For each neighbor of the current node:
       - Calculate the cost to reach the neighbor.
       - Create a new path by adding the neighbor to the current path.
       - Store the new path for further exploration.
  4. **Sort Paths**:
     + Sort all generated paths based on their estimated total cost.

#### 5. ****Repeat Until Goal is Found****

* Continue expanding paths and checking for the goal until it is reached or no paths remain.

#### 6. ****Return the Result****

* Once the goal is found, return the path taken and the total cost.

**Program:**

% Facts: Define the graph connections and their costs

edge(a, b, 1). % cost to go from a to b is 1

edge(a, c, 4). % cost to go from a to c is 4

edge(b, d, 2). % cost to go from b to d is 2

edge(b, e, 5). % cost to go from b to e is 5

edge(c, e, 1). % cost to go from c to e is 1

edge(d, g, 3). % cost to go from d to g is 3

edge(e, g, 2). % cost to go from e to g is 2

% Define the heuristic function for the nodes (estimates cost to goal)

heuristic(a, 7).

heuristic(b, 6).

heuristic(c, 4).

heuristic(d, 3).

heuristic(e, 2).

heuristic(g, 0).

% Best First Search Algorithm

best\_first\_search(Start, Goal, Path, Cost) :-

best\_first([[Start]], Goal, Path, Cost).

best\_first([[Goal | Path] | \_], Goal, [Goal | Path], 0).

best\_first([CurrentPath | OtherPaths], Goal, FinalPath, TotalCost) :-

CurrentPath = [Current | \_],

findall([Next, Current | CurrentPath],

(edge(Current, Next, StepCost), \+ member(Next, CurrentPath),

heuristic(Next, H), F is StepCost + H),

NewPaths),

append(OtherPaths, NewPaths, PathsWithCosts),

sort\_paths(PathsWithCosts, SortedPaths),

best\_first(SortedPaths, Goal, FinalPath, Cost),

TotalCost is Cost + StepCost.

% Sorting paths based on the estimated cost (F)

sort\_paths(Paths, Sorted) :-

map\_list\_to\_pairs(path\_cost, Paths, Pairs),

keysort(Pairs, SortedPairs),

pairs\_values(SortedPairs, Sorted).

path\_cost([Node, Path], F) :-

length(Path, Cost), % Length of the path is used as the cost

heuristic(Node, H), % Get the heuristic cost

F is Cost + H.

% Example Query: To find the best path from a to g

% ?- best\_first\_search(a, g, Path, Cost).

**OUTPUT:**

?- best\_first\_search(a, g, Path, Cost).

Path = [g, e, b, a],

Cost = 8.

**Result:**

The above Prolog program has been executed successfully.

**12.Write a Prolog Program for medical Diagnosis.**

**Aim:**

To Write a Prolog Program for medical Diagnosis.

**Algorithm:**

 **Define Symptoms and Diseases**: Represent diseases and their associated symptoms using facts.

 **User Input**: Collect symptoms from the user.

 **Diagnosis Rules**: Define rules that match symptoms to diseases.

 **Inference Mechanism**: Use Prolog's inference engine to derive the disease based on the symptoms.

 **Output Diagnosis**: Present the possible diagnosis to the user.

**Program:**

% Facts: Define symptoms associated with diseases

disease(flu, [fever, cough, sore\_throat, fatigue]).

disease(cold, [cough, runny\_nose, sore\_throat]).

disease(allergy, [sneezing, runny\_nose, itchy\_eyes]).

disease(pneumonia, [fever, cough, difficulty\_breathing]).

disease(strep\_throat, [sore\_throat, fever, headache]).

disease(migraine, [headache, nausea, sensitivity\_to\_light]).

% Rule: Diagnose based on symptoms

diagnose(Disease) :-

findall(Symptom, symptom(Symptom), Symptoms),

disease(Disease, DiseaseSymptoms),

subset(DiseaseSymptoms, Symptoms).

% Rule: Collect symptoms from the user

symptom(Symptom) :-

write('Do you have the symptom: '), write(Symptom), write('? (yes/no) '),

read(Response),

(Response == yes -> true; fail).

% Example Query: To start the diagnosis

% ?- diagnose(Disease).

**OUTPUT:**

Do you have the symptom: fever? (yes/no)

Disease = flu.

**Result:**

The above Prolog program has been executed successfully.